From Scenarios to Code: An Air Traffic Control Case Study

Jon Whittle
QSS Group/NASA Ames Research Center
jonathw@email.arc.nasa.gov

Richard Kwan
Raytheon/ NASA Ames Research Center
rkwan@mail.arc.nasa.gov

Jyoti Saboo
Foothill College/ NASA Ames Research Center
jsaboo@email.arc.nasa.gov

Abstract

Two high profile workshops at OOPSLA and ICSE, an IEEE Computer article by David Harel and a growing number of research papers have all suggested algorithms that translate scenarios of a system's behavior into state machines. One of the uses of such algorithms is in the transition from requirements scenarios to component design. To date, however, there has been no real evaluation of these algorithms on a realistic case study. In this paper, we do exactly that for the algorithm presented in [10]. Our case study is a component of an air traffic advisory system developed at NASA Ames Research Center.

1 Introduction

There has been a lot of interest recently in the possible role of algorithms that generate state machines automatically from scenarios of intended system behavior — witness, for example, successful workshops at the ICSE02 and OOPSLA01 conferences. A scenario is a trace of an individual execution of a (software) artifact [9]. Scenarios are widely used because they describe concrete interactions and are therefore easier for customers and domain experts to use than an abstract model. Many popular software processes advocate the development of scenarios as an initial software design activity. These scenarios are then used as a starting point to develop more detailed designs, e.g., in the form of state machines. An obvious question to ask is whether this transition can be partially automated. In fact, Harel raised this question in his original paper on statecharts [2] and researchers are now beginning to investigate it (e.g., [4, 5]). To date, however, the focus has been on developing algorithms to translate from scenarios to state machines. The transition is essentially from a *global* scenario-based view (in which interactions between all system components are considered) to *local* component-based views (in which a state machine is given for each component as a precursor to implementation). To the authors' knowledge, there have been no significant case studies in using these algorithms to translate from requirements scenarios to state machines. In this paper, we do exactly that — we apply the algorithm from [10] to the weather control logic subsystem of CTAS (Center TRACON Automation System) which is under development at NASA Ames Research Center. The objective of this study was to assess whether it is possible to use scenario-to-state machine algorithms (ss-algorithms) to reliably develop models of a distributed system.

2 Background

A scenario can be thought of as a particular path through the (intended or actual) behavior of a system. Scenarios can be either exemplary – in which one concrete path is described – or complete, in which a more abstract representation describing multiple concrete instances is given. Scenarios typically involve multiple components of a system. Scenario-to-state machine algorithms can be used on both complete and exemplary scenarios. Their application to exemplary scenarios is generally useful for exploring an incomplete set of requirements to validate or extend it. In contrast, complete scenarios can be used to fully specify a system and complete state machines can be derived from them. In this paper, we will consider the complete case. In addition, we will assume that scenarios are given as UML sequence diagrams and our translation algorithm [10] will output UML statecharts.

The major challenge for ss-algorithms lies in the fact that each scenario is usually written in isolation from the others. To obtain a local model of each component, the scenarios have to be weaved together in such a way that only behavior relevant to that component is extracted and merged together. Most ss-algorithms have as their basis a translation in which messages received by a component in a scenario are considered as trigger events in the component's statechart. Similarly, messages sent to a component are considered as actions for that component (see figure 1[1]). What distinguishes different ss-algorithms is in the way they identify same states in different scenarios and hence weave the scenarios. [7] weaves scenarios based only on a common prefix in the messages and hence cannot merge states that are the same but do not stem from the same sequence of messages. SCED [4] applies merging based on the names of the actions but this does not allow the same action to have a different effect in different states. [5] and others use special state labels to explicitly identify points in different scenarios. [10] allows the user to give more declarative constraints (in the form of message pre/post-conditions) from which state identities can be derived. In this case study, we will use the ss-algorithm from [10] extended with the state labels feature. State labels fit this particular problem well because the scenarios were well defined.

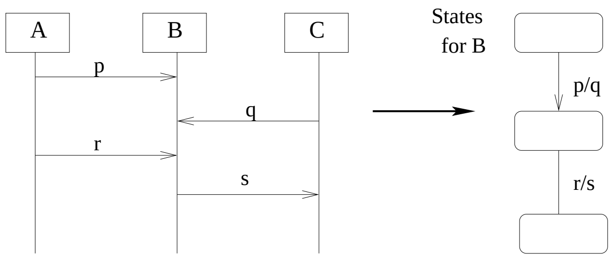

Figure 1. The basic idea underlying most ss-algorithms

3 The Case Study

CTAS (Center TRACON Automation System) [1] is a set of tools designed to help air traffic controllers manage the increasingly complex air traffic flows at large airports. The project began in 1991 and prototypes are now deployed at Denver and Dallas/Fort Worth airports. Extensions to the core CTAS system are constantly being integrated and incorporate the latest developments from research into air traffic control systems. Figure 2 gives an overview of the software architecture for CTAS. CTAS consists of a set of advisory tools and a set of processes that support these tools. In Figure 2, CM is the Communications Manager which handles all communications between the various advisory tools and support processes. TS is the Trajectory Synthesizer which generates 4D trajectories and ETAs that all CTAS tools depend on. RA (Route Analyzer) generates all possible future routes for an aircraft. PFS (Profile Selector) assigns runways for approaching aircraft. PGUI and TGUI are graphical user interfaces. WDAD is a script that is responsible for gathering weather data files from hosts and making them available on the CTAS network file system. WDPD is responsible for converting raw weather files provided via WDAD into binary weather files usable by CTAS.

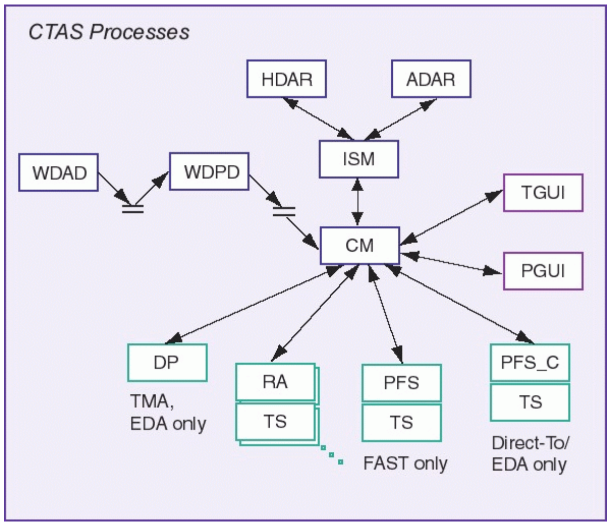

Figure 2. CTAS Architecture Description (taken from `http://www.ctas.arc.nasa.gov`)

One of the most crucial parts of an air traffic control system is the subsystem that deals with weather data. Adverse weather conditions can grind an entire traffic control system to a halt, so it is imperative that each of the components of CTAS is notified of weather forecast updates. The subject of the case study is the subsystem of CTAS that deals with weather data updates. The top-level requirement of this system is that every client that uses weather data should be notified of a weather update and all clients should begin using the updated weather data at the same time. The logic that implements this subsystem is defined by ten pages of English textual requirements. For each weather update, a Weather Cycle is invoked to update all weather aware clients. Similarly, the Weather Cycle is invoked to provide weather data to new clients. The Weather Cycle is split into two subcycles – one for the overall control of the update, and one to keep track of the relative states of the clients. Each requirement describes the behavior of the CM according to the current stage in the cycle, for example:

```
2.8.9  The CM should perform the following actions
       when the Weather Cycle status is 'post-
       initializing' and the newly connected
       weather-aware client has responded no to
       the CTAS_USE_NEW_WTHR messages
       (i.e., wthr_status = FAILED_USE)
```

[1] As usual, `a/b` denotes that `a` is a trigger event and `b` is the action carried out in response to the trigger.

```
(a) it should set the Weather Cycle status to 'done';
(b) it should enable the F2 weather control panel
    ''set'' button
(c) it should send a CM_CLOSE_CONNECTION message to
    the newly connected client
```

4 Objectives

The complete set of requirements describes a state machine expressing the weather update logic component of CTAS, where each requirement describes a partial path through the state machine (i.e., a partial scenario). These partial scenarios also overlap — the requirements designer wrote down each requirement without regard for how it interacts with other requirements.

The code that implements the weather update logic had already been implemented manually in C. The objective of the case study was to reproduce this code directly from the requirements scenarios. The personnel on the case study consisted of a software developer from the CTAS team (not the original developer of the code), one of the researchers who developed the ss-algorithm [10] and a student. The requirements were translated into UML sequence diagrams from which state machines were automatically generated using the tool from [10]. The commercial tool Rational Rose RealTime [3] was then used to generate C++ code from the state machines. This code was integrated into the existing system and tested against the original, manually developed weather control logic code. Twenty of these requirements were written as UML sequence diagrams. In general, the researcher and student came up with the sequence diagrams and then iterated with the CTAS engineer. The other main role of the CTAS engineer was to integrate the code generated from the statecharts into the existing CTAS system and perform testing on it. To summarize the results, this exercise showed that it is possible to generate code directly from UML sequence diagrams. This code passed all test cases. Figure 3 gives the sequence diagram for the requirement 2.8.9 above. Figure 4 shows a portion of the statechart generated for CM. It includes 2.8.9.

5 Observations

5.1 Expressing the requirements as sequence diagrams

All ss-algorithms use some form of sequence chart as their input language — either UML sequence diagrams, high-level message sequence charts (hMSCs) or similar. Sequence diagrams are an attractive choice because of their simplicity. For added expressivity, however, we used an extended version of sequence diagrams with explicit state labels. For this case study, the requirements are already

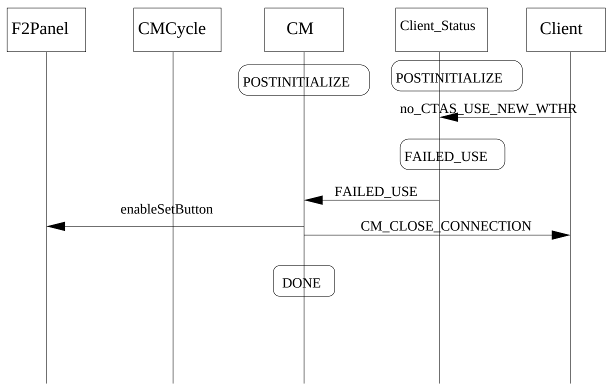

Figure 3. Scenario 2.8.9

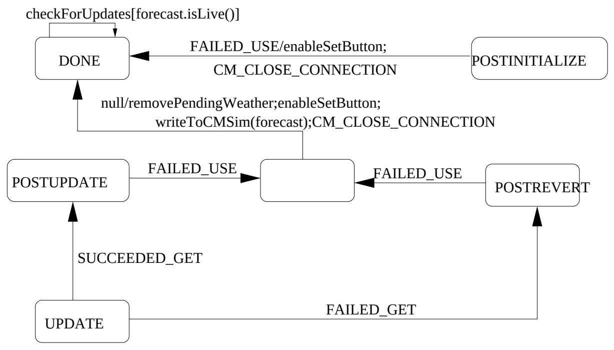
checkForUpdates[forecast.isLive()]
DONE
FAILED_USE/enableSetButton;
CM_CLOSE_CONNECTION
POSTINITIALIZE
null/removePendingWeather;enableSetButton;
writeToCMSim(forecast);CM_CLOSE_CONNECTION
POSTUPDATE
FAILED_USE
FAILED_USE
POSTREVERT
SUCCEEDED_GET
UPDATE
FAILED_GET

Figure 4. Part of the generated statechart

state-oriented — states are used both as preconditions to scenarios and within scenarios to synchronize the states of a subsystem (e.g., client or panel).

Initially, the authors began modeling the requirements directly as a set of state machines. This turned out to be a surprisingly difficult exercise, however. The CM has a number of states corresponding to the current state of the Weather Cycle: unknown, preinitializing, initializing, postinitializing, preupdating, updating, postupdating, postreverting and done. Each weather-aware client has these states in addition to states describing whether or not the client has received new weather data and whether or not it is able to use this new weather data (succeeded_get, failed_get, succeeded_use, failed_use). In order to capture the requirements correctly, the states of the CM and of the clients must be tightly coupled together. These couplings are distributed across the entire requirements document and so capturing them directly as state machines is a time consuming and error-prone task. In contrast, translating the twenty requirements into sequence diagrams took about one hour of clerical work.

There were, however, a number of issues that could not easily be expressed as sequence diagrams (or any other sequence chart notation). Many of the messages in the requirements are universal or existential messages — that is, the scenario is dependent on receipt of a message from *all* (alternatively, *any one of*) the instances of a classifier. It is unclear how to represent this on a sequence diagram. Of course, this could be done by attaching a textual note but the difficulty is representing this information in such a way that a synthesis algorithm can generate appropriate states for it. One possibility is to use UML's constraint language, OCL ([8]). The requirement that "all connected weather-aware clients have responded yes to the CTAS_USE_NEW_WTHR" message could be written as `Client.allInstances()-> forall(i | receive(CM, yes_CTAS_USE_NEW_WTHR, i))` where $receive(i, m, j)$ denotes the receipt of message m from instance j by instance i. The problem with using OCL is that the user is given too much freedom to specify arbitrary constraints that may or may not be relevant to synthesis. The synthesis algorithm would need to look for OCL patterns to trigger the generation of the appropriate state machine transitions. This would rely on the user specifying the constraint in the form given above rather than a semantically equivalent one. An alternative is to include special textual (or graphical) syntax to denote existentiality or universality. In general, however, this may be inadequate if there turns out to be a large number of special cases like this one. In this case study, it turned out to be sufficient to encode the universality (or existentiality) directly in the messages — i.e., to send a message to all clients, a new message $send\_to\_all$ is created. This is adequate but, in general, it seems beneficial to be able to express this kind of information explicitly.

Sequence diagrams are currently inadequate for describing generalized scenarios. A generalized scenario is one that describes a set of possible scenarios rather than a single trace. We give two examples of generalized scenarios here. Firstly, one may wish to specify that a message can be sent or received at any point during an interaction. In hMSCs, this could be expressed using coregions. Alternatively, a group of messages may be order independent — i.e., there is a scenario for each possible ordering of the messages and all are valid. Clearly, describing each of these scenarios individually is time consuming. For hMSCs, ordering of messages can be left underspecified using coregions. Many generalized scenarios (but not all) can be expressed using hMSCs. However, current ss-algorithms cannot generate appropriate states for generalized scenarios. Hence, the use of these algorithms can lead to mismatches between the input sequences and the generated state machines. One possible approach to including generalized scenarios in ss-algorithms is to generate hierarchical states. For example, the requirement that a component can receive messages m_1, m_2, m_3 in any order can be succinctly expressed using orthogonal states and the join operator (which impedes progress until all its input transitions have fired) — see Figure 5.

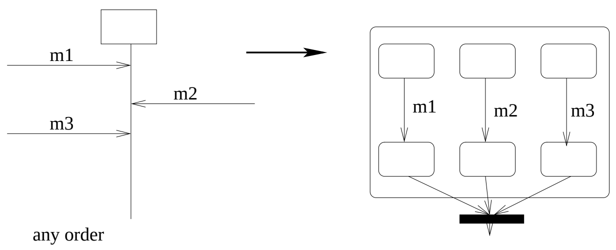

Figure 5. Translating Generalized Scenarios

The use of state labels alone is also insufficient in general. The continuation from one sequence diagram to another can be expressed by state labels (see Figure 3) but a better solution would have been to use hMSCs. hMSCs include a notation for specifying how individual sequence charts connect using continuation, iteration and choice operators. On the other hand, hMSCs are more complex than sequence diagrams. Note, also, that state labels are still needed in hMSCs for connecting points in the middle of a sequence chart.

In total, twenty requirements were translated into sequence diagrams. The procedure was mostly painless, the main detail being in deciding on which components to represent in the diagrams. For example, we decided to represent Client_Status explicitly as a component but could alternatively have represented it as a variable. One suggestion for ss-algorithms is to allow the user to specify how a component should be translated — currently, all algorithms will generate a state machine for each component in the sequences. However, it may be useful to be able to give a component explicitly in the sequence diagram but then implement it in the design or code as a variable. For details on the structure of the sequence diagrams, see section 6.

5.2 Translating into statecharts

The twenty sequence diagrams were translated into statecharts using the algorithm in [10]. The requirements centred around the behavior of CM and Client_Status so statecharts were generated only for these components. There were not enough messages to make it meaningful to generate a statechart for F2Panel. For Client, no information was specified as to when it should send its messages so a highly non-deterministic state machine could have been generated and possibly used to generate test cases, but this was not done (test cases were developed manually).

The major barrier in the use of ss-algorithms currently is in avoiding over-generalization of the input scenarios. Many forms of generalization are possible – e.g., generalizing a concrete instance to a variable, generalizing a message to be sent to *all* instances of a class – but most of these probably cause more problems than they solve because it is very difficult to identify over-generalizations in a generated state machine (it requires thoroughly understanding the state machine and failure to do so may result in bugs). There is one form of generalization that is crucial for ss-algorithms to work well, however – that of merging same states from different scenarios. If no attempt is made to merge states, the result is a state machine with essentially one branch for each scenario that contains a lot of redundant states and duplication of transitions. A variety of approaches to this problem have been proposed. Our algorithm [10] takes a heuristic approach – states can be merged if they have an identical sequence of transitions above some given length leading into and out of them.

Figure 4 illustrates the benefit of merging. There are three transitions with the trigger FAILED_USE. In two cases, the paths following FAILED_USE in the corresponding scenarios are the same and so are merged. In the third case, merging is not appropriate because the following path is an alternative. In general, merging in this way produces a much more concise and readable state machine. Research still needs to be done, however, on the best ways to introduce merging. Using our heuristic approach, over-zealous merging can still occur — in this case study, a small number of over-generalizations had to be modified by hand.

The other major observation made during application of the ss-algorithm was that there were some “hidden dependencies” in the sequence diagrams. Consider Figure 1. According to the semantics of the algorithm, component C is only dependent on messages q and s, and hence, only q and s will appear in C’s state machine. However, it may be that message q can only be sent once p has been received by B (alternatively, sending q may be independent of p). We term such a dependency “hidden”. Our ss-algorithm has no way to detect hidden dependencies and the resulting state machines will not synchronize components correctly in such cases. Our solution was to add an explicit handshake between components B and C. This solution worked well but tool support for detecting hidden dependencies would have been useful. Early research in this area has begun [6] but we did not try out these techniques on this case study.

5.3 Generating code

Once state machines were generated for the CM and Client_Status components, the Rational Rose RealTime tool was used to generate C++ code. This code was then integrated into the existing CTAS code base. This was generally a straightforward process. Each action in the transitions corresponded to a method in the original code base. In the original design, the weather update subsystem of CM is polled from its environment every two seconds. Once polled, the module runs to completion, finishing by sending a message to one or more clients. At this point, the weather update subsystem yields the focus of control. In order to facilitate this, the state machines generated were given additional triggers. Yielding of focus of control can occur in a given number of states. When control is regained, execution should continue from the exit state. These states were given a trigger that is called from the environment. This mechanism gives the same behavior as the two second polling in the original design.

At the point of integrating the code, a small number of misunderstandings were discovered in the way the scenarios had been written. These modifications were made to the state machine and code re-generated. Unfortunately, in the time crush that was now ensuing, changes were not reflected in the scenarios. This leads us to the conclusion that a crucial part of this technology should be to automatically maintain consistency between sequence diagrams and state machines. This could be done with a “backwards direction” algorithm that keeps track of the changes to the state machine and suggests corresponding changes in the scenarios. Although there has been some research on this topic, a good solution has not yet been found.

Rose RealTime generates code that includes its own run-time services library. The CTAS team were not ready to accept a third party library such as this because of concerns about reliability. As a result, the code was generated using RealTime *passive classes* which do not require a run-time library. State machines for passive classes, however, have a synchronous model of execution — all triggers are just function calls so there are no event queues on state machines. The main problem with this is that the sequence diagrams were written assuming an asynchronous model. As a result, the state machine did not exhibit the expected behavior. This mismatch necessitated some minor changes to the state machines.

6 Results Summary

Tables 1 and 2 give some basic statistics on the problem scenarios and the generated artifacts for the portion of the weather control logic subsystem considered in this paper. If we had used hMSCs instead of sequence diagrams extended with state labels, we could have avoided inserting most of the state labels. The RoseRT comments referred to in the table are special comments introduced by the RoseRT code generator to keep track of which code is auto-generated for the purposes of round-trip engineering.

Whilst the authors believe this case study to provide evidence that transitioning mostly automatically from scenarios to code is possible, it should be noted that this case study has a number of characteristics that may or may not be shared with other examples. Firstly, the requirements

General data	
# of sequence diagrams	17
# of components per diagram	3-6
Messages data	
total # of messages	75
# of messages per diagram	2-12
average # of messages per diagram	4.4
# of messages that don't appear in state machines	0
State labels data	
# of state labels	45
# of state labels avoided by hMSCs	36

Table 1. Sequence diagram statistics

	CM	Client_Status
# states	15	11
# LOC generated by RoseRT	986	640
# LOC w/out RoseRT comments	672	404

Table 2. State machine statistics

were very well developed. They represent a complete and consistent view of the system and hence, there were very few and only minor iterations in developing the scenarios. In a case where the ss-algorithm was being used to help develop the requirements, additional results would be observed. Secondly, the length of the sequence diagrams (in terms of number of messages) turned out to be quite small on average. This is because the requirements were already well structured. This might not be the case for examples in which the requirements were vague. Thirdly, it is interesting to note that the requirements already identify most of the states (e.g., postinitialize) that appear in the state machine. Clearly, the scenarios were written from a state-based perspective which may have made the transition to state machines easier. In summary, though, we believe that this case study represents a realistic example that provides interesting results to the researchers in this field.

7 Conclusions

This case study has shown that it is possible to generate code mostly automatically from scenarios of the intended behavior. Although there are still some hurdles to overcome, we believe it would have been possible for the CTAS engineer to carry out this process independently. Interestingly, the engineer could very easily understand the sequence diagrams but had trouble understanding the statecharts generated. The hardest part of the process was actually in integrating the generated code into the existing CTAS system. In particular, the messages had to be mapped to existing method calls and the previous implementation made use of variables to represent the weather cycle that became obsolete but were still used by other components.

In generating statecharts from scenarios, simple algorithms work the best. The more advanced features of our particular algorithm were not used and we would conjecture that other complex mechanisms designed to avoid overgeneralization would cause too much confusion. Current ss-algorithms are generally good enough to do a reasonable job. The main area that is not currently well supported is in maintaining the consistency of the different viewpoints under modifications to the generated artifacts.

References

[1] D. Denery, H. Erzberger, T. Davis, S. Green, and B. McNally. Challenges of air traffic management research: Analysis, simulation and field test. In *AIAA Guidance, Navigation and Control Conference*, 1997.

[2] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[3] *Rational Rose RealTime*. Rational Software Corporation, Cupertino, CA, 2002.

[4] T. Systä. Incremental construction of dynamic models for object oriented software systems. *Journal of Object Oriented Programming*, 13(5):18–27, 2000.

[5] S. Uchitel and J. Kramer. A workbench for synthesizing behavior models from scenarios. In *Proceedings of the 23rd IEEE International Conference on Software Engineering (ICSE01)*, Toronto, Canada, 2001.

[6] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 9th European Software Engineering Conference (ESEC01)*, Vienna, Austria, 2001.

[7] A. van Lamsweerde. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998.

[8] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[9] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. *IEEE Software*, pages 34–45, March/April 1998.

[10] J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland, June 2000.